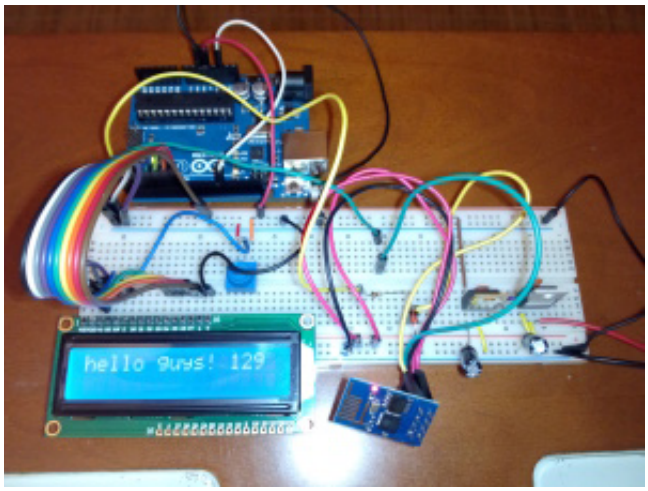
SISTEMAS O.R.P

Blog sobre informática, electrónica, robótica y otros temas

Programando un Arduino remotamente con el módulo ESP8266



Una de mis viejas aspiraciones cuando construyo robots es poder programarlos sin tener que recogerlos, enchufarles un cable usb o un programador ICSP y volverlos a dejar en su sitio.

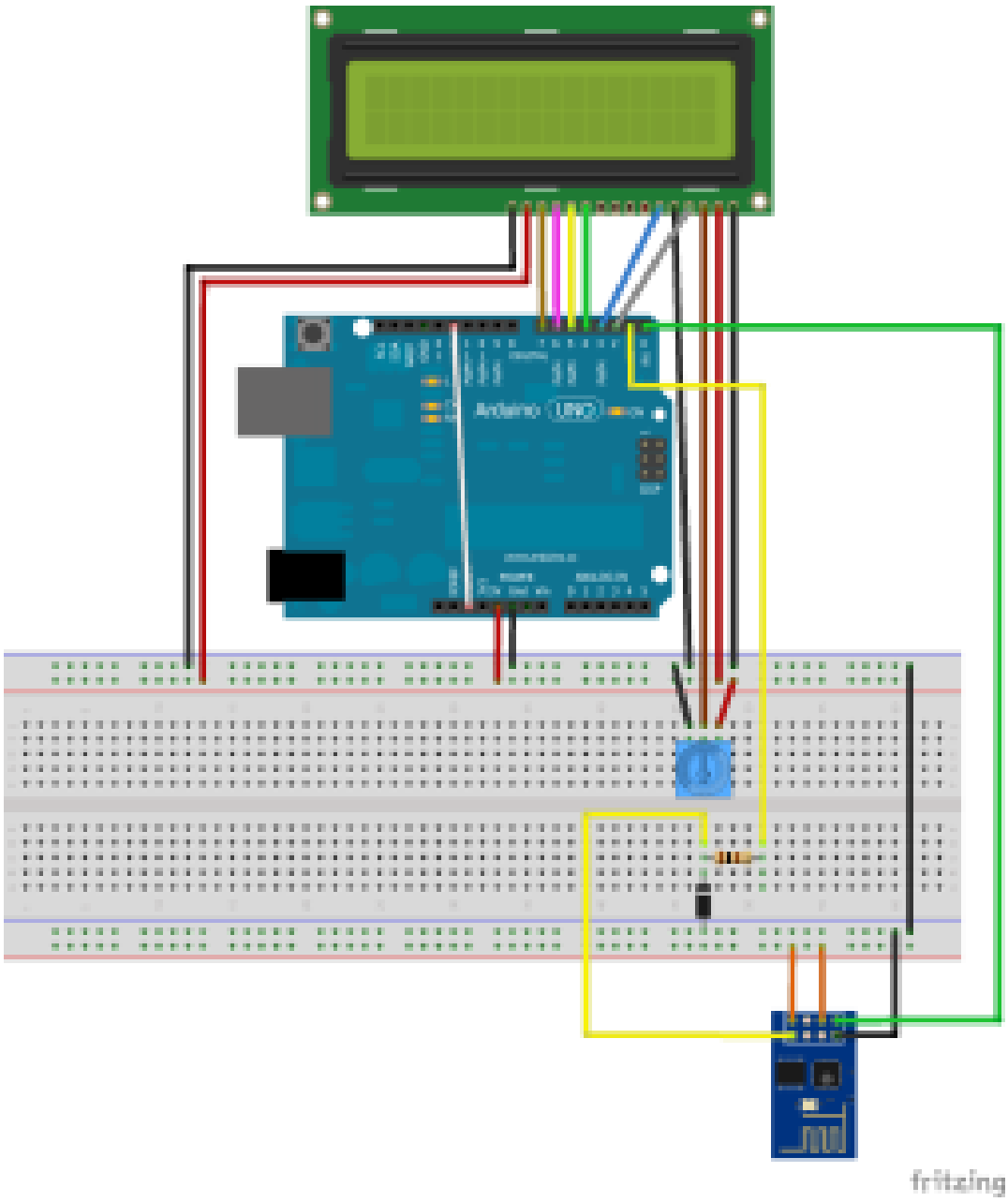
En este artículo explicaré cómo con un Arduino UNO y un módulo ESP8266 (llamado WI07C) se puede programar un sketch de Arduino en la placa sin tener que estar cerca de esta, todo mediante wifi y sockets tcp/ip.

Descripción general

Como se puede ver en el vídeo tenemos un Arduino UNO conectado a un display HD44780 y a un módulo wifi ESP8266. Arduino envía cadenas *hello world* al servidor sockettest que está escuchando por el puerto 49000. Se modifica el código Arduino en el IDE poniendo *hello folks*, se compila y el fichero .hex generado (se puede ver donde está activandolo en Archivo/Preferencias/compilación) se copia a la carpeta del programa en python. Cuando Arduino recibe un comando *reboot* se conecta al servidor python por el puerto 50000 entrando en modo de datos, acto seguido se reinicia y empieza la programación remota de Arduino. El proceso se reliza dos veces en el vídeo.

Lo que se aprovecha es el método que tiene Arduino para programarse, ya que usando el puerto serie después de un reset se puede programar un Arduino si se sigue el protocolo STK500 implementado en el bootloader.

Esquema de conexiones



El Arduino y el display HD44780 se alimentan a 5 voltios, el módulo ESP8266 se alimenta a 3,3 voltios. Como el pin rx del módulo wifi sólo puede funcionar a 3,3V se usa un diodo zener de 3,3 voltios junto con una resistencia de 100 Ω.

En las placas Arduino, el pin de reset del microcontrolador está conectado a una resistencia y esta a su vez está conectada a VCC, con lo que para el microcontrolador el pin está a nivel alto. Cuando se pulsa el botón de reset lo que se hace es derivar la corriente hacia masa (ya que el bóton está conectado a esta) y el microcontrolador, al estar a nivel bajo el pin de reset, realiza un reseteo. Cuando el microcontrolador arranca, todos los pines están configurados como entradas (alta impedancia) y por eso no le afecta que el pin de reset esté conectado directamente al pin 12. Si se configura el pin 12 como salida y luego se conecta a masa (nivel bajo o LOW) se provoca el mismo efecto que si se pulsase el botón de reset, además, al existir la resistencia anteriormente mencionada, no hay que preocuparse de que se produzca un cortocircuito al unir VCC con masa (GND).

Sketch de Arduino

```
// Copyright (C) 2014 SISTEMAS O.R.P.
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.

#include <LiquidCrystal.h>

#define SEARCH_AFTER_RESET "ready"
#define INTRO "\r\n"

#define AP_NORMAL "SISTEMASORP"
#define PWD_NORMAL "mypassword"
#define HOST_NORMAL "192.168.1.33"
#define PORT_NORMAL "49000"

#define AP_BOOTLOADER "SISTEMASORP"
#define PWD_BOOTLOADER "mypassword"
#define HOST_BOOTLOADER "192.168.1.33"
#define PORT_BOOTLOADER "50000"


boolean ok = false;
int counter = 0;
LiquidCrystal lcd(2, 3, 4, 5, 6, 7);
//--------------------------------------------------------------------------------
/*
Parameters:
  show: The string to be printed.
Returns:
  Nothing.
Description:
  It clears the LCD and print the string.
*/
void print_lcd(String show)
{
  lcd.clear();
  lcd.print(show);
}
//--------------------------------------------------------------------------------
/*
Parameters:
  command: The string to send.
  timeout: The maximum time in milliseconds the function can be running before a time o
  wait_for1: The search string when the command succeeded.
  wait_for1: The search string when the command failed.
Returns:
  The string contained in wait_for1, wait_for2 or the string TIMEOUT.
Description:
```

```
  It sends the command trough the serial port and waits for one of the expected strings
*/
String send(String command, int timeout, String wait_for1, String wait_for2)
{
  unsigned long time = millis();
  String received = "";

  Serial.print(command);
  Serial.print(INTRO);

  while(millis() < (time + timeout))
  {
    if(Serial.available() > 0)
    {
      received.concat(char(Serial.read()));
      if(received.indexOf(wait_for1) > -1)
      {
        return wait_for1;
      }
      else if(received.indexOf(wait_for2) > -1)
      {
        return wait_for2;
      }
    }
  }

  return "TIMEOUT";
}
//------------------------------------------------------------------------------
/*
Parameters:
  wait_for: The search string.
  timeout: The maximum time in milliseconds the function can be running before a time o
Returns:
  True if the string was found, otherwise False.
Description:
  It waits for the expected string.
*/
boolean look_for(String wait_for, int timeout)
{
  unsigned long time = millis();
  String received = "";

  while(millis() < (time + timeout))
  {
    if(Serial.available() > 0)
    {
      received.concat(Serial.readString());
      if(received.indexOf(wait_for) > -1)
      {
        return true;
      }
    }
  }

  return false;
}
//------------------------------------------------------------------------------
/*
Parameters:
  command: The string to send.
```

```
  timeout: The maximum time in milliseconds the function can be running before a timeou
  wait_for1: The search string when the command succeeded.
  wait_for1: The search string when the command failed.
Returns:
  True if the wait_for1 string was found, otherwise False.
Description:
  It sends the command trough the serial port and waits for one of the expected strings
*/
boolean send_command(String command, int timeout, String wait_for1, String wait_for2)
{
  String state;

  state = send(command, timeout, wait_for1, wait_for2);
  if(state == wait_for1)
  {
    return true;
  }
  else if(state == wait_for2)
  {
    // do something on error
  }
  else
  {
    // do something on timeout
  }

  return false;
}
//------------------------------------------------------------------------------------
/*
Parameters:
  Nothing
Returns:
  True if all commands were successfully, otherwise False.
Description:
  It initializes the module, joins to the access point and connects to the server.
*/
boolean init_commands()
{
  print_lcd("Changing mode");
  if(send_command("AT+CWMODE=1", 5000, "OK", "ERROR"))
  {
    print_lcd("Resetting module");
    if (send_command("AT+RST", 5000, SEARCH_AFTER_RESET, "ERROR"))
    {
      print_lcd("Joining AP");

      String cwjap = "AT+CWJAP=\"";
      cwjap += AP_NORMAL;
      cwjap += "\",\"";
      cwjap += PWD_NORMAL;
      cwjap += "\"";
      if (send_command(cwjap, 20000, "OK", "FAIL"))
        if (send_command("AT+CIPMUX=0", 2000, "OK", "ERROR"))
          if (send_command("AT+CIPMODE=0", 2000, "OK", "ERROR"))
          {
            print_lcd("Connecting host");

            String cipstart = "AT+CIPSTART=\"TCP\",\"";
            cipstart += HOST_NORMAL;
            cipstart += "\",";
```

```
181             cipstart += PORT_NORMAL;
182             if (send_command(cipstart, 5000, "OK", "ERROR"))
183               return true;
184           }
185     }
186   }
187
188   return false;
189 }
190 //------------------------------------------------------------------------------
191 /*
192 Parameters:
193   Nothing
194 Returns:
195   True if all commands were successfully, otherwise False.
196 Description:
197   It initializes the module, joins to the access point, connects to the server and star
198 */
199 boolean boot_commands()
200 {
201   print_lcd("Joining AP");
202
203   String cwjap = "AT+CWJAP=\"";
204   cwjap += AP_BOOTLOADER;
205   cwjap += "\",\"";
206   cwjap += PWD_BOOTLOADER;
207   cwjap += "\"";
208   if (send_command(cwjap, 20000, "OK", "FAIL"))
209     if (send_command("AT+CIPMUX=0", 2000, "OK", "ERROR"))
210       if (send_command("AT+CIPMODE=1", 2000, "OK", "ERROR"))
211       {
212         print_lcd("Connecting host");
213
214         String cipstart = "AT+CIPSTART=\"TCP\",\"";
215         cipstart += HOST_BOOTLOADER;
216         cipstart += "\",";
217         cipstart += PORT_BOOTLOADER;
218         if (send_command(cipstart, 5000, "OK", "ERROR"))
219           if (send_command("AT+CIPSEND", 2000, ">", "ERROR"))
220           {
221             print_lcd("Init protocol");
222             if (send_command("hello", 2000, "welcome", "error"))
223               if (send_command("Arduino_remote_example.cpp.hex", 2000, "ok", "error"))
224                 return true;
225           }
226       }
227
228   return false;
229 }
230 //------------------------------------------------------------------------------
231 /*
232 Parameters:
233   Nothing
234 Returns:
235   True if all commands were successfully, otherwise False.
236 Description:
237   It sends a string to the remote host and show it in the display.
238 */
239 boolean test()
240 {
241   String command = "AT+CIPSEND=";
```

```
  String to_send = "hello guys! ";
  to_send += counter;
  command += to_send.length() + 2;

  if (send_command(command, 2000, ">", "ERROR"))
    if (send_command(to_send + "\r\n", 2000, "OK", "ERROR"))
    {
      lcd.clear();
      lcd.print(to_send);
      counter++;
      return true;
    }

  return false;
}
//------------------------------------------------------------------------------
void setup()
{
  pinMode(13, OUTPUT);
  Serial.begin(115200);
  lcd.begin(16, 2);
  lcd.print("Init WI07C...");

  // Remove any garbage from the RX buffer
  delay(3000);
  while(Serial.available() > 0) Serial.read();

  ok = init_commands();
}
//------------------------------------------------------------------------------
void loop()
{
  if(ok)
  {
    digitalWrite(13, HIGH);
    ok = test();
    if(ok && look_for("reboot", 5000))
    {
      if(boot_commands())
      {
        pinMode(12, OUTPUT);
        digitalWrite(12, LOW);
        for(;;);
      }
      else
      {
        ok = false;
      }
    }
  }
  else
  {
    digitalWrite(13, LOW);
    lcd.clear();
    lcd.print("Error sending");
    lcd.setCursor(0, 1);
    lcd.print("AT commands");
    for(;;);
  }
}
```

Lo que hace el skecth de Arduino es:

- Inicializa el puerto serie a 115200 bps.
- Elimina los caracteres que hubiera en el buffer de entrada del puerto serie.
- Inicializa el módulo wifi en modo estación, lo resetea, se conecta al punto de acceso *normal*, configura las conexiones como simples en modo normal y se conecta al servidor normal por el puerto 49000. Si hubiese algún fallo en algún punto pararía la ejecución y lo indicaría en el display.
- Envia una cadena de texto y un número consecutivo tanto al servidor como al display. Si hubiese algún fallo en algún punto pararía la ejecución y lo indicaría en el display.
- Si entre el envío de las cadenas de texto se recibe una cadena *reboot* entonces resetea el módulo, se conecta al punto de acceso *bootloader*, configura las conexiones como simples en modo normal y se conecta al servidor de programación por el puerto 50000. Acto seguido envía una cadena *hello* y espera a recibir una cadena *welcome*, si ha sido así, entonces envía el nombre del fichero .hex con el que quiere ser programado el Arduino y espera a recibir una cadena *ok*, momento en el cual configura el pin 12 como salida y lo pone a nivel bajo, conectándolo a masa y provocando un reset en el Arduino. Si hubiese algún fallo en algún punto pararía la ejecución y lo indicaría en el display.

Aquí cabe destacar cómo funciona el sistema de reseteo: Justo después del reseteo, el pin de TX de Arduino está a nivel bajo durante un tiempo, lo que provoca que el módulo wifi vea eso como un byte 0, que enviará a través de la conexión TCP/IP. El servidor de programación aprovechará esta circunstancia para saber cuando ha empezado el reseteo e iniciar el protocolo STK500. El bootloader de Arduino entra en acción después del reseteo y espera a recibir ordenes del protocolo STK500, si las recibe actúa en consecuencia, si no, ejecuta el programa principal.

Servidor de programación

```
#!/usr/bin/env python

# Copyright (C) 2014 SISTEMAS O.R.P.
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.

import sys
import binascii
import struct
import select
import socket
```

```
23  import errno
24
25  MAX_TIMEOUT = 500
26  SUCCESS = "success"
27  FAILED = "failed"
28  PORT = 50000
29  #------------------------------------------------------------------------------
30  '''
31  Class containing the data from non-contiguous memory allocations
32  '''
33  class Data:
34      def __init__(self, begin, data):
35          self.begin = begin
36          self.data = data
37          self.count = len(data)
38  #------------------------------------------------------------------------------
39  '''
40  Parameters:
41      line: The line to parse
42  Returns:
43      The size of data. The address of data. The type of data. The line checksum. True if
44  Description:
45      It parses a line from the .hex file.
46  '''
47  def parse_line(line):
48          ok = False
49          size = int(line[1:3], 16)
50          address = int(line[3:7], 16)
51          type = int(line[7:9], 16)
52          next_index = (9 + size * 2)
53          data = binascii.a2b_hex(line[9:next_index])
54          checksum = int(line[next_index:], 16)
55
56          #checking if checksum is correct
57          sum = size + (address >> 8) + (address & 0xFF) + type
58          for byte in data:
59              sum += ord(byte)
60
61          if (~(sum & 0xFF) + 1) & 0xFF == checksum:
62              ok = True
63
64          return (size, address, type, data, checksum, ok)
65  #------------------------------------------------------------------------------
66  '''
67  Parameters:
68      chunks: An array with different chunks of data.
69      path: The path to the .hex file to read
70  Returns:
71      True if the reading was successfully, otherwise False.
72  Description:
73      It reads a .hex file and stores the data in memory.
74  '''
75  def read_hex_file(chunks, path):
76      try:
77          file = open(path, 'r')
78      except IOError:
79          print "Hex file not loaded"
80          return False
81      line = file.readline()
82      if line[0] != ':':
83          print "The file seems to be a not valid .hex file"
```

```
84          file.close()
85          return False
86
87      size, address, type, data, checksum, ok = parse_line(line.strip())
88      if not ok:
89          print "The checksum in line 1 is wrong"
90          file.close()
91          return False
92
93      chunks.append(Data(address, data))
94
95      # Read the other lines
96      index = 0
97      count = 2
98      for line in file:
99          size, address, type, data, checksum, ok = parse_line(line.strip())
100         if not ok:
101             print "The checksum in line", count, "is wrong"
102             file.close()
103             return False
104
105         if chunks[index].begin + chunks[index].count == address:
106             chunks[index].count += size
107             for code in data:
108                 chunks[index].data += code
109         else:
110             chunks.append(Data(address, data))
111             index += 1
112         count += 1
113
114     return True
115 #---------------------------------------------------------------------------
116 '''
117 Parameters:
118     None
119 Returns:
120     The server socket
121 Description:
122     It opens a server socket at the specified port and listens to connections.
123 '''
124 def init_server():
125     server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
126     server.bind(('', PORT))
127     server.listen(1)
128     return server
129 #---------------------------------------------------------------------------
130 '''
131 Parameters:
132     cli: The client socket
133     response: The search string
134     timeout: The maximum time in milliseconds the function can be running before a time
135 Returns:
136     True if the string was found, otherwise False. The received string.
137 Description:
138     It waits for the expected string.
139 '''
140 def wait_for(cli, response, timeout):
141     inputs = [cli]
142     received = ""
143     milliseconds = 0
144     while milliseconds < timeout:
```

```
145         rlist, wlist, xlist = select.select(inputs,[],[], 0.001)
146         if len(rlist) > 0:
147             received += cli.recv(1)
148             if response in received:
149                 return True, received
150         milliseconds += 1
151 
152     return False, received
153 #------------------------------------------------------------------------------
154 '''
155 Parameters:
156     cli: The client socket
157     timeout: The maximum time in milliseconds the function can be running before a time
158     length: The number of bytes to receive.
159 Returns:
160     True if the string has the required length, otherwise False. The received string.
161 Description:
162     It waits for the required length of bytes.
163 '''
164 def return_data(cli, timeout, length = 1):
165     inputs = [cli]
166     received = ""
167     milliseconds = 0
168     while milliseconds < timeout:
169         rlist, wlist, xlist = select.select(inputs,[],[], 0.001)
170         if len(rlist) > 0:
171             received = cli.recv(length)
172             return True, received
173         milliseconds += 1
174 
175     return False, received
176 #------------------------------------------------------------------------------
177 '''
178 Parameters:
179     cli: The client socket
180 Returns:
181     True if the string was found, otherwise False
182 Description:
183     It waits for the acknowledge string.
184 '''
185 def acknowledge(cli):
186     if wait_for(cli, "\x14\x10", MAX_TIMEOUT)[0]: #STK_INSYNC, STK_OK
187         print SUCCESS
188         return True
189     else:
190         print FAILED
191         return False
192 #------------------------------------------------------------------------------
193 '''
194 Parameters:
195     chunks: An array with different chunks of data.
196     cli: The client socket
197 Returns:
198     Nothing
199 Description:
200     It starts the STK500 protocol to program the data at their respective memory addres
201 '''
202 def program_process(chunks, cli):
203     print "Connection to Arduino bootloader:",
204 
205     counter = 0
```

```
206     cli.send("\x30\x20") #STK_GET_SYNCH, SYNC_CRC_EOP
207     if not acknowledge(cli):
208         return
209 
210     print "Enter in programming mode:",
211     cli.send("\x50\x20") #STK_ENTER_PROGMODE, SYNC_CRC_EOP
212     if not acknowledge(cli):
213         return
214 
215     print "Read device signature:",
216     cli.send("\x75\x20") #STK_READ_SIGN, SYNC_CRC_EOP
217     if wait_for(cli, "\x14", MAX_TIMEOUT)[0]: #STK_INSYNC
218         ok,received = return_data(cli, MAX_TIMEOUT, 3)
219         print binascii.b2a_hex(received)
220         if not wait_for(cli, "\x10", MAX_TIMEOUT)[0]: #STK_INSYNC
221             print FAILED
222             return
223     else:
224         print FAILED
225         return
226 
227     for chunk in chunks:
228         total = chunk.count
229         if total > 0: #avoid the last block (the last line of .hex file)
230             current_page = chunk.begin
231             pages = total / 0x80
232             index = 0
233 
234             for page in range(pages):
235                 print "Load memory address",current_page,":",
236                 cli.send(struct.pack("<BHB", 0x55, current_page, 0x20)) #STK_LOAD_ADDRE
237                 if not acknowledge(cli):
238                     return
239 
240                 print "Program memory address:",
241                 cli.send("\x64\x00\x80\x46" + chunk.data[index:index + 0x80] + "\x20")
242                 if not acknowledge(cli):
243                     return
244                 current_page += 0x40
245                 total -= 0x80
246                 index += 0x80
247 
248             if total > 0:
249                 print "Load memory address",current_page,":",
250                 cli.send(struct.pack("<BHB", 0x55, current_page, 0x20)) #STK_LOAD_ADDRE
251                 if not acknowledge(cli):
252                     return
253 
254                 print "Program memory address:",
255                 cli.send(struct.pack(">BHB", 0x64, total, 0x20) + chunk.data[index:inde
256                 if not acknowledge(cli):
257                     return
258 
259     print "Leave programming mode:",
260     cli.send("\x51\x20") #STK_LEAVE_PROGMODE, SYNC_CRC_EOP
261     acknowledge(cli)
262 #------------------------------------------------------------------------------
263 def main():
264     print "Arduino remote programmer 2014 (c) SISTEMAS O.R.P"
265 
266     print "Listen to connections"
```

```
267     ser = init_server()
268     inputs = [ser]
269 
270     while True:
271         rlist, wlist, xlist = select.select(inputs,[],[])
272         for s in rlist:
273             if s == ser:
274                 cli, addr = s.accept()
275                 print addr[0], "connected"
276                 # It assures the connection is for programming an Arduino and not other
277                 if wait_for(cli, "hello", 5000)[0]:
278                     cli.send("welcome")
279                     ok, received = wait_for(cli, "hex", 5000)
280                     if ok:
281                         chunks = []
282                         print "Read hex file", received.strip()
283                         if read_hex_file(chunks, received.strip()):
284                             cli.send("ok")
285                             # Wait for the byte '0' sent by Arduino after resetting
286                             if wait_for(cli, "\x00", MAX_TIMEOUT)[0]:
287                                 program_process(chunks, cli)
288                         else:
289                             cli.send("error")
290                 cli.close()
291                 print "Listen to connections"
292 #------------------------------------------------------------------------------
293 if __name__ == "__main__":
294     main()
```

Lo que hace el servidor de programación es:

- Crea un socket que escuche por el puerto 50000
- Cuando un cliente de conecta espera a la cadena *hello*, si la recibe reponde con una cadena *welcome*.
- Espera a que el cliente le envíe un nombre de fichero .hex. Trata de abrir el fichero en el mismo directorio y lo lee interpretando todas las líneas para guardar los datos del programa en memoria. Si todo va bien envía una cadena *ok*.
- Espera a recibir el byte 0, y cuando lo recibe empieza el protocolo STK500 para comunicarse con el bootloader de Arduino y programarlo. Básicamente lo que hace es entrar en modo de programación, indicarle a que páginas quiere acceder y enviar los datos de cada página, así hasta que ha enviado todos los datos del programa, después sale del modo de programación cerrando la conexión

Aquí cabe destacar que cuando se cierra la conexión TCP/IP con el cliente (ya sea por un error o porque el proceso de programación ya ha terminado), el módulo ESP8266 sale del modo de datos automáticamente y no es necesario que el nuevo sketch tenga que enviarle la cadena de escape +++ para poder entrar otra vez en modo de comandos.

Conclusiones

Espero que esto os sirva para que en vuestros proyectos podais programar remotamente vuestros Arduinos a través de una red local o Internet.

Habría que tener en cuenta que el proceso de actualización puede quedarse a medias si la conexión a la red wifi es lenta y provocaría que el programa no se ejecutara correctamente. Así que que una mejora sería acoplar un chip aparte que reprogramase el Arduino por ICSP con un programa preestablecido en caso de que detectase que la programación no fue finalizada correctamente.

Comentar que la versión de firmware del módulo ESP8266 que he usado es la 0019000902 y el Arduino es un UNO. Ambos funcionan a 115200 bps, pero si quisierais utilizar otras velocidades (en otros Arduinos el bootloader configura el puerto serie a 19200 o 57600 bps) habría que cargar un firmware que lo permitiese, como por ejemplo la versión 0.922 de electrodragon y su comando *AT+CIOBAUD*.